Une approche gloutonne pour établir la singleton consistance d'arc

Stephane Cardon, Christophe Lecoutre

Une approche gloutonne pour établir la singleton consistance d'arc

Stéphane Cardon **Christophe Lecoutre**

CRIL - CNRS FRE 2499
Université d'artois
rue de l'université, SP 16
62307 Lens cedex, France

{cardon,lecoutre}@cril.univ-artois.fr

Résumé

Dans cet article, nous proposons une nouvelle approche pour établir la singleton consistance d'arc (SAC) d'un réseau de contraintes. Tandis que le principe des algorithmes SAC existants consiste à réaliser un parcours en largeur d'abord jusqu'à une profondeur égale à 1, le principe des deux algorithmes que nous introduisons consiste à réaliser plusieurs exécutions d'une recherche gloutonne (telle que la consistance d'arc soit maintenue à chaque étape). Il s'agit d'une illustration originale d'inférence (i.e. établir la singleton consistance d'arc) par la recherche. Utiliser une approche gloutonne permet de bénéficier de l'incrémentalité de la consistance d'arc, d'apprendre des informations pertinentes à partir des conflits et de, potentiellement, trouver des solutions pendant le processus d'inférence. De plus, les complexités temporelle et spatiale sont tout à fait compétitives.

Abstract

In this paper, we propose a new approach to establish Singleton Arc Consistency (SAC) on constraint networks. While the principle of existing SAC algorithms involves performing a breadth-first search up to a depth equal to 1, the principle of the two algorithms introduced in this paper involves performing several runs of a greedy search (where at each step, arc consistency is maintained). It is then an original illustration of applying inference (i.e. establishing singleton arc consistency) by search. Using a greedy search allows benefiting from the incrementality of arc consistency, learning relevant information from conflicts and, potentially finding solution(s) during the inference process. Furthermore, both space and time complexities are quite competitive.

1 Introduction

L'inférence et la recherche constituent deux catégories de méthodes et techniques dans le domaine de la satisfaction de contraintes [12]. D'une part, l'inférence est utilisée pour transformer une instance d'un problème en une forme équivalente qui est soit directement exploitable afin de démontrer la satisfiabilité ou l'insatisfiabilité de cette instance, soit plus simple dans le but d'être traité par un algorithme de recherche. L'inférence consiste donc à modifier un réseau de contraintes en employant des méthodes structurelles telles que l'élimination de variables et le "tree clustering", ou des méthodes de filtrage basées sur des propriétés telles que la consistance d'arc ou celle de chemin. D'autre part, la recherche consiste à parcourir l'espace délimité par le domaine de toutes les variables du problème. La recherche peut être systématique et complète à l'aide d'un parcours en largeur d'abord ou en profondeur d'abord avec retour-arrières, ou stochastique et incomplète grâce à une exploration locale et des heuristiques aléatoires.

L'un des algorithmes de recherche systématique parmi les plus populaires pour résoudre les instances du Problème de Satisfaction de Contraintes, appelées instances CSP (Constraint Satisfaction Problem) ou encore réseaux de contraintes, est l'algorithme MAC (Maintaining Arc Consistency) [18]. MAC entrelace inférence et recherche puisqu'à chaque étape d'une recherche en profondeur d'abord avec retour-arrières, une consistance locale appelée la consistance d'arc (AC pour Arc Consistency) [15] est maintenue. Cependant, depuis l'introduction de consistances plus fortes telles que la consistance de chemin max-restreinte (Max-RPC pour Max-restricted path consis-

tency) [10] et la singleton consistance d'arc (SAC pour Singleton Arc Consistency) [11], la question de pouvoir utiliser en pratique de telles consistances, au lieu de la consistance d'arc, avant ou pendant la recherche, se pose. En particulier, on peut noter récemment un fort engouement pour les singleton consistances et, plus particulièrement, pour SAC, comme cela est illustré par les travaux récents [11, 17, 1, 3, 4]. Un réseau de contraintes est dit singleton arc consistant si et seulement si après avoir réduit le domaine d'une variable à un singleton (c'est à dire effectué une assignation) et établi la consistance d'arc, le réseau ne se trouve pas être trivialement inconsistant par la détection d'un domaine vide.

Plusieurs algorithmes pour établir la singleton consistance d'arc ont été récemment proposés. Le premier d'entre eux, SAC-1 [11], a la bonne propriété de n'utiliser aucune structure de données particulière. Sa complexité spatiale est ainsi donnée par celle de l'algorithme de consistance d'arc sous-jacent. Cependant, sa complexité temporelle dans le pire des cas est assez élevée puisqu'elle est en O(mn^2d^4) où n représente le nombre de variables, d la taille du plus grand domaine et m le nombre de contraintes. Le deuxième algorithme proposé, SAC-2 [1], admet la même complexité temporelle dans le pire des cas que SAC-1, mais, en évitant certains tests inutiles de singleton consistance, il s'avère en pratique plus efficace (en temps). Cette amélioration est au prix d'une complexité spatiale en O(n^2d^2). Ensuite, un algorithme, appelé SAC-OPT, ayant une complexité temporelle optimale en O(mnd^3) a été proposé dans [3]. Néanmoins, sa complexité spatiale en O(mnd^2) le rend inutilisable sur des réseaux de contraintes de taille importante. Finalement, l'algorithme SAC-SDS [4] a été élaboré à partir de SAC-OPT en relâchant l'optimalité de la complexité temporelle, pour une meilleure complexité spatiale. Ses complexités en temps et en espace sont en O(mnd^4) et en O(n^2d^2) respectivement.

Dans cet article, nous proposons deux nouveaux algorithmes, appelés SAC-3 et SAC-3+, permettant d'établir la singleton consistance d'arc. Alors que les algorithmes présentés ci-dessus effectuent une recherche en largeur d'abord jusqu'à une profondeur égale à 1, les deux nouveaux algorithmes effectuent plusieurs essais de recherche gloutonne (telle qu'à chaque étape, la consistance d'arc est maintenue). A la différence de SAC-3+, SAC-3 n'enregistre pas l'environnement de chaque essai.

Nous avons identifié plusieurs avantages à cette approche :

- l'exigence spatiale est nulle pour SAC-3 et limitée pour SAC-3+,
- les deux algorithmes tirent avantage de l'incrémentalité de la consistance d'arc,
- l'utilisation d'une recherche gloutonne permet l'apprentissage d'informations pertinentes à partir des conflits rencontrés,
- il est possible qu'une solution soit trouvée lors de l'établissement de la consistance,
- les deux algorithmes sont compétitifs en terme d'efficacité temporelle.

Plus précisément, la bonne complexité spatiale de ces deux algorithmes permet leur utilisation sur des réseaux de contraintes de taille importante. En particulier, SAC-3 a la même complexité en espace que l'algorithme de consistance d'arc sous-jacent. Par ailleurs, l'incrémentalité de la consistance d'arc est exploitée lorsqu'une recherche gloutonne maintenant la consistance d'arc est utilisée. Cela signifie qu'établir la consistance d'arc itérativement sur un espace de recherche de plus en plus réduit est moins pénalisant que de répéter l'établissement de la consistance d'arc sur l'espace de recherche original. De plus, lorsqu'un essai de recherche gloutonne se termine par un échec, un "nogood" peut-être enregistré et/ou la cause de cet échec pris en compte. Il est également possible qu'une ou plusieurs solutions puissent être trouvées lorsque un ou plusieurs essais de recherche gloutonne aboutissent à un succès. Finalement, si l'instance est sous-contrainte ou contient une partie sous-contrainte importante, la complexité temporelle devient très intéressante.

Cet article est organisé comme suit. Après quelques préliminaires (Section 2) et le survol des algorithmes SAC existants (Section 3), nous introduisons le principe d'une approche gloutonne pour établir la singleton consistance d'arc (Section 4). Ensuite, les deux algorithmes que nous proposons et qui sont basés sur ce principe de recherche gloutonne sont introduits (Sections 5 et 6). Les résultats obtenus lors de nos expérimentations sont alors présentés (Section 7) avant de conclure.

2 Préliminaires

Définition 1 *Un réseau de contraintes P est un couple $(\mathscr{X}, \mathscr{C})$ où :*

- *$\mathscr{X} = \{X_1, \ldots, X_n\}$ est un ensemble fini de n variables tel que chaque variable X_i possède un domaine $dom(X_i)$ représentant l'ensemble des valeurs pouvant être affectées à X_i,*
- *$\mathscr{C} = \{C_1, \ldots, C_m\}$ est un ensemble fini de m contraintes tel que chaque contrainte C_j correspond à une relation $rel(C_j)$ représentant l'ensemble des tuples autorisés pour les variables $vars(C_j) \subseteq \mathscr{X}$ liées par la contrainte C_j.*

Une solution est une assignation de valeurs à l'ensemble des variables telle que toutes les contraintes soient satisfaites. Un réseau de contraintes est satisfiable lorsqu'il admet au moins une solution. Le problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem), qui consiste à déterminer si un réseau de contraintes donné est satisfiable, est NP-complet. Un réseau de contraintes est

également appelé instance CSP. Dans cet article, résoudre une instance revient soit à trouver une solution, soit à démontrer qu'elle est insatisfiable.

Les méthodes de recherche complètes (dites aussi systématiques) utilisent, en règle générale, un algorithme de recherche en profondeur d'abord avec gestion de retour-arrières, où à chaque étape de la recherche, une assignation de variable est effectuée suivie par un processus de filtrage appelé propagation de contraintes. Dans la plupart des cas, les algorithmes de propagation de contraintes qui sont basés sur certaines propriétés des réseaux de contraintes telles que la consistance d'arc [15], éliminent des valeurs qui ne peuvent apparaître dans aucune solution.

Définition 2 *Soient $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes, $C \in \mathscr{C}$, $X \in vars(C)$ et $a \in dom(X)$. (X, a) est dit arc consistant pour C si et seulement si il existe un support de (X, a) dans C, c'est-à-dire, un tuple $t \in rel(C)$ tel que $t[X] = a$. P est dit arc consistant si et seulement si $\forall X \in \mathscr{X}$, dom(X) $\neq \emptyset$ et $\forall C \in \mathscr{C}$, $\forall X \in vars(C)$, $\forall a \in dom(X)$, (X, a) est arc consistant pour C.*

AC(P) représente le réseau de contraintes obtenu après avoir établi la consistance d'arc sur le réseau donné P. AC(P) correspond donc au réseau P pour lequel toutes les valeurs qui ne sont pas arc consistantes ont été éliminées. Remarquons qu'une valeur correspondra habituellement ici à un couple (X,a) où $X \in \mathscr{X}$ et $a \in dom(X)$. Si il existe une variable ayant un domaine vide dans AC(P), noté AC(P) = $\bot$, alors P est clairement insatisfiable. $P|_S$ où $S \subset \{X = a | X \in \mathscr{X} \wedge a \in dom(X)\}$ est le réseau de contraintes obtenu depuis P en restreignant le domaine de X au singleton $\{a\}$ pour toute assignation $X = a \in S$.

Définition 3 *Soient $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes, $X \in \mathscr{X}$ et $a \in dom(X)$. (X, a) est dit singleton arc consistant si et seulement si AC($P|_{X=a}$) $\neq \bot$. P est dit singleton arc consistant si et seulement si $\forall X \in \mathscr{X}$, dom(X) $\neq \emptyset$ et $\forall a \in dom(X)$, (X, a) est singleton arc consistant.*

$\mathscr{X}$ sera appelé le domaine du réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$. On notera aussi $(X, a) \in P$ (respectivement, $(X, a) \notin P$) si et seulement si $X \in \mathscr{X}$ et $a \in dom(X)$ (respectivement, $a \notin dom(X)$).

3 Survol des algorithmes SAC

Le premier algorithme proposé pour établir la singleton consistance d'arc est SAC-1 [11]. Le principe de cet algorithme est de vérifier la singleton consistance d'arc, variable après variable, jusqu'à ce qu'un point fixe soit atteint. Pour chaque variable, toute valeur détectée singleton arc inconsistante est éliminée du domaine de cette variable. Les complexités en espace et en temps dans le pire des cas de SAC-1 sont respectivement O(md) et O(mn^2d^4) où n représente le nombre de variable, d la taille du plus grand domaine et m le nombre de contraintes.

Un deuxième algorithme, appelé SAC-2, a été proposé par Bartak [1]. L'idée est de ne vérifier (à nouveau) la singleton consistance d'arc d'une valeur (Y,b), après la suppression d'une valeur (X,a), que si (X,a) ne supporte pas (Y,b), c'est-à-dire (X,a) n'appartient pas à AC($P|_{Y=b}$). Cet algorithme évite donc des vérifications inutiles en enregistrant, pour chaque valeur, l'ensemble des valeurs supportées par celle-ci. Comme cela est pressenti et comme le montrent les expérimentations de Bartak [1], SAC-2 offre une amélioration conséquente du temps de résolution en pratique par rapport à SAC-1. Les complexités en espace et temps dans le pire des cas sont, respectivement, O(n^2d^2) et O(mn^2d^4).

Bessière et Debruyne [3] ont remarqué que SAC-2 ne présente pas une amélioration théorique de de la complexité dans le pire des cas à cause de la vérification systématique depuis le départ (i.e. sans tenir compte du travail qui a déjà été effectué) de la singleton consistance d'arc d'une valeur lorsque celle-ci doit à nouveau être effectuée. En d'autres termes, SAC-2 n'exploite pas l'incrémentalité de la consistance d'arc. Un algorithme de consistance d'arc est dit incrémental si sa complexité en temps dans le pire des cas est la même qu'il soit appliqué une fois sur un réseau donné P ou qu'il soit appliqué n fois sur P tel qu'entre deux exécutions consécutives, au moins une valeur ait été supprimée. Tous les algorithmes connus de consistance d'arc sont incrémentaux. Pour bénéficier de l'incrémentalité de la consistance d'arc, Bessière et Debruyne [3] ont proposé un nouvel algorithme, appelé SAC-OPT dans [4], qui duplique le réseau de contraintes original nd fois, chacun étant dédié à une valeur (X,a) de l'instance. Le principe est alors d'utiliser le réseau dédié à une valeur (X,a) lorsque la consistance d'arc de cette valeur doit être vérifiée à nouveau. Les complexités en espace et en temps dans le pire des cas de SAC-OPT sont, respectivement, en O(mnd^2) et O(mnd^3), ce qui représente la meilleure complexité en temps qui peut être escomptée pour un algorithme établissant la singleton consistance d'arc [3].

Finalement, à partir de l'observation qu'une complexité spatiale en O(mnd^2) ne permet pas d'utiliser un algorithme tel que SAC-OPT sur des réseaux de contraintes de taille importante, Bessière et Debruyne [4] ont proposé un algorithme, appelé SAC-SDS, qui représente un compromis entre complexité temporelle et complexité spatiale. Par rapport à chaque valeur, seul le domaine (appelé SAC-support) est enregistré ainsi qu'une liste de propagation utilisée pour la consistance d'arc. En retour, les structures de données utilisées pour établir la consistance d'arc ne sont plus dédiées mais partagées. Une étude expérimentale menée sur des instances aléatoires a mis en évidence les bonnes per-

formances de cet algorithme. Les complexités en espace et en temps dans le pire des cas de SAC-SDS sont, respectivement, en O(n^2d^2) et O(mnd^4).

Afin d'être exhaustif, notons également que dans le cadre des réseaux de contraintes pour lesquels chaque domaine est défini par un intervalle de valeurs[1], la restriction de la singleton consistance d'arc aux bornes du domaine de chaque variable est appelé 3B-consistance [14]. Un algorithme optimal pour établir la 3B-consistance a été proposé par Bordeaux et al. [6]. Enfin, en ordonnancement, une technique de "shaving" articulée autour de la singleton consistance d'arc aux bornes a été proposée dans [16].

4 Une approche gloutonne

Tous les algorithmes présentés ci-dessus utilise une recherche en largeur d'abord jusqu'à une profondeur égale à 1. Chaque branche (de taille 1) de cette recherche correspond à une vérification de la singleton arc consistance d'une valeur et permet la suppression de celle-ci si une inconsistance est trouvée (après établissement de la consistance d'arc). Pour illustrer cette approche "classique", considérons la figure 1 où les différentes valeurs d'un réseau sont testées l'une après l'autre. Par exemple, on peut observer sur la figure que (X,a) est détecté singleton arc consistant tandis que (X,b) ne l'est pas (ce qui est symbolisé par la croix).

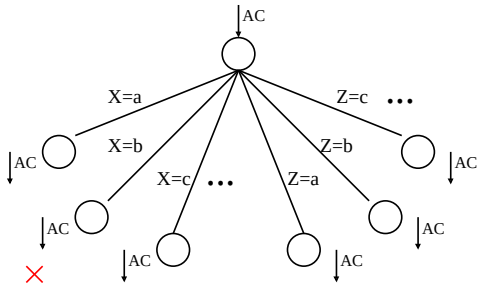

FIG. 1 – Approche classique

Une alternative est de vérifier la singleton arc consistance d'une valeur dans la continuité des vérifications précédentes. En d'autres termes, nous pouvons essayer de construire moins de branches de plus grande taille en utilisant une recherche gloutonne (au niveau de laquelle, à chaque étape, la consistance d'arc est maintenue). Tant qu'aucune inconsistance n'est détectée au niveau d'une branche, celle-ci est étendue. Lorsqu'une inconsistance est détectée, soit il s'agit d'une branche de taille 0 (une seule assignation menant directement à un échec lorsque la consistance d'arc est établie) et alors une valeur est détectée inconsistante et peut donc être éliminée, soit toutes les assignations associées à la branche correspondent à une valeur singleton arc consistante, exceptée la dernière. Pour illustrer cette approche "gloutonne", considérons la figure 2. Une première branche est construite en sélectionnant (Y,a) puis, cette valeur étant singleton arc consistante, (X,b). Un échec étant constaté, il est nécessaire de reconsidérer (X,b) car il n'est pas possible de déterminer si cette valeur est singleton arc consistante ou non. Par contre, la singleton arc consistance de (Y,a) a été démontrée. Une seconde branche est construite en sélectionnant (X,b). Cette valeur étant détectée singleton arc inconsistante, elle peut être éliminée. Plus tard, une branche est construite, et en considérant un réseau comportant uniquement 3 variables, les 3 (assignations de) valeurs (Y,c), (Z,a) et (X,a) correspondent à une solution. Cela signifie que non seulement la singleton arc consistance des 3 valeurs est démontrée mais qu'une solution est trouvée de manière opportuniste.

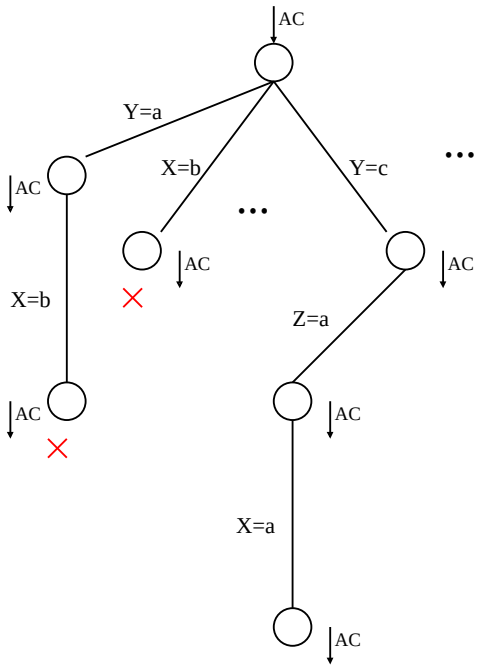

FIG. 2 – Approche gloutonne

L'approche gloutonne exploite la proposition suivante.

Proposition 1 *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes et soit $S \subset \{X = a | X \in \mathscr{X} \wedge a \in dom(X)\}$. Si AC($P|_S$) $\neq \perp$ alors tout couple (X,a), tel que $X \in \mathscr{X}$ et dom(X) = $\{a\}$ dans AC($P|_S$), est singleton arc consistant.*

Preuve Si AC($P|_S$) $\neq \perp$, alors clairement tout élément $X = a \in S$ est singleton arc consistant. Il s'agit d'une conséquence de la monotonie de l'arc consistance : si un réseau de contraintes P est arc consistant alors tout réseau obtenu à partir de P par suppression d'un certain nombre de contraintes l'est aussi. On peut observer qu'il existe

[1] Les instances CSP numériques appartiennent à ce cadre.

aussi certaines valeurs (Y,b) telles que $Y \in \mathscr{X}$ et dom(Y) = $\{b\}$ dans AC($P|_S$) où $Y = b \notin S$. Ces valeurs sont aussi clairement arc consistantes. □

Comme mentionné dans la preuve précédente, quelques valeurs peuvent-être détectées singleton arc consistantes lors de la vérification de la singleton arc consistance d'une autre valeur. La proposition 1 peut alors être vue comme une généralisation de la propriété 3 définie dans [9], et se trouve aussi liée à l'exploitation des variables singleton-valuées de [19].

Bien que le but premier de notre approche est d'exploiter l'aspect incrémental de l'arc consistance, d'autre propriétés intéressantes peuvent-être observées. En effet, en utilisant une recherche gloutonne, une solution peut-être trouvée de façon opportuniste et un apprentissage à partir des conflits effectué : par exemple, des "no-good" peuvent être enregistrés et/ou le poids de certaines contraintes incrémenté [7].

5 SAC-3

Dans cette section, nous donnons la description d'un premier algorithme utilisant une recherche gloutonne dans le but d'établir la singleton consistance d'arc d'un réseau. Cette description est donnée dans le contexte d'utilisation d'un algorithme de consistance d'arc sous-jacent à gros grain (tel que AC3 [15], AC2001/3.1 [5, 21] ou AC3.2/3.3 [13]) avec un schéma de propagation orienté-variable.

D'abord, introduisons quelques notations. Si $P = (\mathscr{X}, \mathscr{C})$, alors AC($P$,$Q$) où $Q \subseteq \mathscr{X}$ revient à établir la consistance d'arc sur P à partir de l'ensemble de propagation Q qui contient des variables dont le domaine a été récemment modifié. Pour une description de l'algorithme AC, nous reportons le lecteur, par exemple, à la fonction $propagateAC$ définie dans [4]. Q_{sac} est l'ensemble des valeurs pour lesquelles la singleton consistance d'arc doit-être vérifiée. Une branche correspond à un ensemble de valeurs qui ont été assignées. Pour tout ensemble de valeurs $S \subseteq \{(X,a) \mid X \in \mathscr{X} \wedge a \in dom(X)\}$, vars($S$) = $\{X|(X,a) \in S\}$. Finalement, une instruction de la forme $P_{before} \leftarrow P$ peut ne pas être considérée comme une duplication du problème. La plupart du temps, cela correspond à conserver ou mettre à jour le domaine d'un réseau de contraintes (et les structures de l'algorithme de consistance d'arc sous-jacent).

L'algorithme 2 commence par établir la consistance d'arc sur le réseau donné. Ensuite, toutes les valeurs sont placées dans la structure Q_{sac} et, afin de vérifier leur singleton consistance d'arc, des branches successives sont créées. Le processus continue jusqu'à atteindre un point fixe.

L'algorithme 1 permet la construction de branches en assignant successivement les variables tout en maintenant la consistance d'arc (ligne 6). Lorsqu'une inconsistance est détectée pour une branche non vide, on replace la dernière valeur dans Q_{sac} (ligne 12) puisqu'aucune information n'est disponible sur la singleton arc consistance ou inconsistance de cette valeur. Si la branche est vide, la valeur doit être supprimée et la consistance d'arc doit être rétablie (lignes 16 à 19). Remarquons qu'aucune inconsistance n'est détectée lorsqu'une solution est trouvée et lorsqu'il n'existe plus aucun autre moyen d'étendre la branche courante.

Algorithm 1 buildBranch()

```
1:  br ← ∅
2:  P_before ← P
3:  consistant ← true
4:  repeat
5:     choisir et éliminer (X,a) ∈ Q_sac tel que X ∉ vars(br)
6:     P ← AC(P|_{X=a},{X})
7:     if P ≠ ⊥ then
8:        ajouter (X,a) à br
9:     else
10:       consistant ← false
11:       if br ≠ ∅ then
12:          ajouter (X,a) à Q_sac
13:    end if
14: until not consistant ∨ (vars(Q_sac) − vars(br) = ∅)
15: P ← P_before
16: if br = ∅ then
17:    éliminer a de dom(X)
18:    P ← AC(P,{X})
19:    Q_sac ← Q_sac − {(Y,b)|(Y,b) ∈ dom(P_before) − dom(P)}
20: end if
```

Algorithm 2 SAC-3($P = (\mathscr{X}, \mathscr{C})$: CSP)

```
1: P ← AC(P,X)
2: repeat
3:    P_before ← P
4:    Q_sac ← {(X,a) | X ∈ X ∧ a ∈ dom(X)}
5:    while Q_sac ≠ ∅ do
6:       buildBranch()
7: until P = P_before
```

Proposition 2 *SAC-3 est un algorithme correct ayant, dans le pire des cas, une complexité spatiale en O(md) et une complexité temporelle en O(bmd^2) où b représente le nombre de branches créées par SAC-3.*

Preuve La correction est issue directement de la Proposition 1. Si SAC-3 utilise un algorithme de consistance d'arc sous-jacent optimal tel que AC2001/3.1, alors la complexité spatiale de l'algorithme est globalement O(md) puisque la complexité spatiale de AC2001/3.1 est en O(md), la structure de donnée Q_{sac} est en O(nd) et chaque branche créée est en O(n). La complexité temporelle est en O(bmd^2) puisqu'en conséquence de l'incrémentalité, chaque construction de branche est en O(md^2). □

Remarquons que b doit inclure les branches "vides" qui correspondent à la détection de valeurs singleton arc inconsistantes. En ce qui concerne les réseaux de contraintes qui

sont initialement singleton arc consistants, le corollaire 1 indique que SAC-3 apparaît comme une alternative intéressante à SAC-OPT et SAC-SDS (admettant alors une complexité temporelle, dans le pire des cas, en O(mnd^3)). Cela suggère aussi que SAC-3 peut sans doute être plus performant que SAC-OPT et SAC-SDS vis à vis des instances (pas nécessairement singleton arc consistantes) contenant une partie sous-contrainte de taille importante comme cela peut l'être pour les applications réelles.

Corollaire 1 *SAC-3 admet une complexité temporelle dans le pire des cas en O(mn^2d^4), mais, lorsqu'il est appliqué à un réseau singleton arc consistant, SAC-3 admet une complexité temporelle dans le meilleur des cas*[2] *en O(md^3) et, dans le pire des cas, en O(mnd^3).*

Preuve Dans le pire des cas, $b = \frac{n^2d^2+nd}{2}$, d'où, nous obtenons O(mn^2d^4). Lorsque l'algorithme est appliqué à une instance initialement singleton arc consistante, le meilleur et le pire des cas correspondent respectivement à des branches de taille maximale et de taille 1 (1 assignation consistante suivie d'une assignation inconsistante). Nous avons donc, respectivement, $b = d$ (toutes les branches mènent à une solution) et $b = nd$ branches. □

6 SAC-3+

Il est possible d'améliorer le comportement de SAC-3 en enregistrant le domaine des réseaux de contraintes obtenus après chaque exécution gloutonne, autrement dit, après chaque branche construite. Lorsqu'une valeur est supprimée, il est alors possible de déterminer quelles branches précédemment créées doivent-être reconsidérées. En effet, si une valeur ne supporte pas une branche, c'est-à-dire, n'appartient pas au domaine associé à la branche, toutes les valeurs de la branche restent singleton arc consistantes. D'autre part, si elle supporte une branche, nous avons à vérifier que la branche reste valide en ré-établissant la consistance d'arc à partir du domaine enregistré. Lorsqu'une branche n'est plus valide, on la supprime. En résumé, SAC-3+ exploite l'incrémentalité de la consistance d'arc à la manière de SAC-3 (en menant des essais de recherche gloutonne) mais également à la manière de SAC-SDS (en évitant de recommencer le travail à partir du départ).

Dans le but de gérer domaines et ensembles de propagations liés aux réseaux de contraintes correspondant aux branches, nous considérons deux tableaux notés P et Q. Pour une branche donnée br, $P[br]$ correspond au réseau de contraintes associé à la branche br (en fait, nous avons seulement besoin d'enregistrer le domaine de ce réseau) alors que $Q[br]$ contient les variables dont le domaine a été

[2] Même si le pire des cas est considéré pour l'algorithme de consistance d'arc sous-jacent.

Algorithm 3 update(set : Ensemble de valeurs)

```
1: Q_sac ← Q_sac − set
2: for chaque branche br ∈ brs do
3:   for chaque valeur (X,a) ∈ set do
4:     if (X,a) ∈ P[br] then
5:       éliminer (X,a) de P[br]
6:       ajouter X à Q[br]
7:     endif
```

Algorithm 4 buildBranch+()

```
1: br ← ∅
2: P_before ← P
3: consistant ← true
4: repeat
5:   choisir et éliminer (X,a) ∈ Q_sac tel que X ∉ vars(br)
6:   P ← AC(P|_{X=a},{X})
7:   if P ≠ ⊥ then
8:     ajouter (X,a) à br
9:     P_store ← P
10:  else
11:    consistant ← false
12:    if br ≠ ∅ then
13:      ajouter (X,a) à Q_sac
14:  end if
15: until not consistant ∨ (vars(Q_sac) − vars(br) = ∅)
16: P ← P_before
17: if br = ∅ then
18:   éliminer a de dom(X)
19:   P ← AC(P,{X})
20:   update({(Y,b)|(Y,b) ∈ dom(P_before) − dom(P)})
21: else
22:   P[br] ← P_store
23:   ajouter br à brs
24: end if
```

Algorithm 5 checkBranches()

```
1: for chaque branche br ∈ brs do
2:   P[br] ← AC(P[br], Q[br])
3:   if P[br] = ⊥ then
4:     Q_sac ← Q_sac ∪ br
5:     éliminer br de brs
6:   end if
7: end for
```

Algorithm 6 SAC-3+($P = (\mathscr{X}, \mathscr{C})$: CSP)

```
1: P ← AC(P,𝒳)
2: brs ← ∅
3: Q_sac ← {(X,a) | X ∈ 𝒳 ∧ a ∈ dom(X)}
4: while Q_sac ≠ ∅ do
5:   while Q_sac ≠ ∅ do
6:     buildBranch+()
7:   checkBranches()
8: end while
```

récemment réduit et qui doivent donc être considérées lors du ré-établissement de la consistance d'arc.

Après avoir établi la consistance d'arc sur le réseau de contraintes donné, l'algorithme 6 crée successivement des branches en appelant la fonction $buildBranch+$. Une fois que la singleton consistance d'arc a été testée pour chaque valeur de Q_{sac}, nous vérifions la validité des branches créées et enregistrées dans brs par appel à la fonction $checkBranches$. Pour chaque branche br, la consistance d'arc est ré-établie pour $P[br]$ (ligne 2 de l'algorithme 5) et dans le cas où un domaine vide est détecté, nous supprimons cette branche et mettons à jour Q_{sac} (lignes 4 et 5).

L'algorithme 4 diffère de l'algorithme 1 sur deux aspects. Premièrement, nous avons besoin d'enregistrer le domaine du réseau correspondant à la branche créée (ligne 22) et d'ajouter cette branche à brs (ligne 23). Remarquons que si la dernière assignation de variable entraîne la détection d'un domaine vide, P_{store} n'est pas mis à jour (ligne 9). Au niveau de l'implantation, $P[br]$ peut-être directement établi (en faisant un retour-arrière d'un pas si nécessaire) sans aucune duplication de domaine. Deuxièmement, après ré-établissement de la consistance d'arc (ligne 19), toutes les valeurs supprimées, incluant celle qui est singleton arc inconsistante (ligne 18), doivent être prises en compte dans le but de mettre à jour l'état de toutes les branches (ligne 20).

Proposition 3 *SAC-3+ est un algorithme correct ayant une complexité spatiale en O($b_{max}nd + md$) et une complexité temporelle en O(bmd^2) où b_{max} représente le nombre maximal de branches enregistrées par SAC-3+ et b représente le nombre de branches créées ou testées par SAC-3+.*

Preuve La correction résulte de la proposition 1 et du fait que, une fois la singleton consistance d'arc testée pour toutes les valeurs et les branches enregistrées, la propriété est vérifiée par appel à $checkBranches$. Conjointement à la complexité spatiale en O(md) de l'algorithme optimal de consistance d'arc sous-jacent, il est nécessaire d'enregistrer le domaine de chaque réseau correspondant aux branches valides qui ont été créées. Comme l'espace requis pour enregistrer un domaine (de réseau) est en O(nd), nous obtenons une complexité spatiale en O($b_{max}nd + md$). □

Remarquons que le corollaire 1 reste valide pour SAC-3+. Toutefois, cet algorithme devrait s'avérer être plus efficace en pratique que SAC-3 puisque SAC-3+ évite de (re)construire des branches lorsque cela n'est pas utile. La complexité spatiale de SAC-3+ est donnée par le corollaire suivant.

Corollaire 2 *SAC-3+ admet une complexité spatiale en O(n^2d^2) dans le pire des cas et en O($nd^2 + md$) dans le meilleur des cas.*

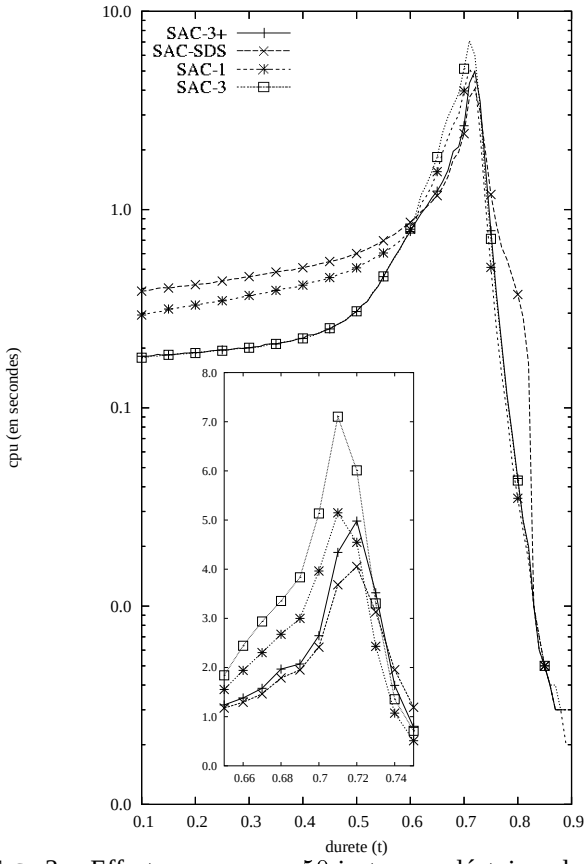

FIG. 3 – Effort moyen pour 50 instances aléatoires de la classe (100,20,0.05,t)

7 Expérimentations

Afin de montrer l'intérêt pratique des algorithmes introduits dans cet article, nous les avons implantés ainsi que les algorithmes SAC-1 et SAC-SDS, ce dernier étant considéré comme l'algorithme le plus efficace actuellement [4]. Nous avons utilisé AC3.2 [13] comme algorithme de consistance d'arc sous-jacent. Nous avons conduit les expérimentations pour différentes classes d'instances aléatoires, académiques et réelles sur un PC Pentium IV 2,4GHz 512Mo sous Linux. Pour chaque instance, nous avons cherché à établir la singleton consistance d'arc sur le réseau correspondant avec les 4 algorithmes mentionnés (et non cherché à maintenir SAC durant la recherche d'une solution). La performance des algorithmes a été mesurée en termes du nombre de tests de singleton consistance d'arc (#scks) et du temps cpu en secondes (cpu). Pour information, nous avons donné, pour chaque instance, le nombre (#X) de valeurs supprimées lorsque SAC est établie sur cette instance (lorsque #X=0, l'instance considérée est alors initialement singleton arc consistante). Pour SAC-3 et SAC-3+, nous donnons aussi (excepté pour les instances aléatoires) le nombre b de branches construites ainsi que leur longueur moyenne g (arrondie à 1 chiffre après la virgule) sous la

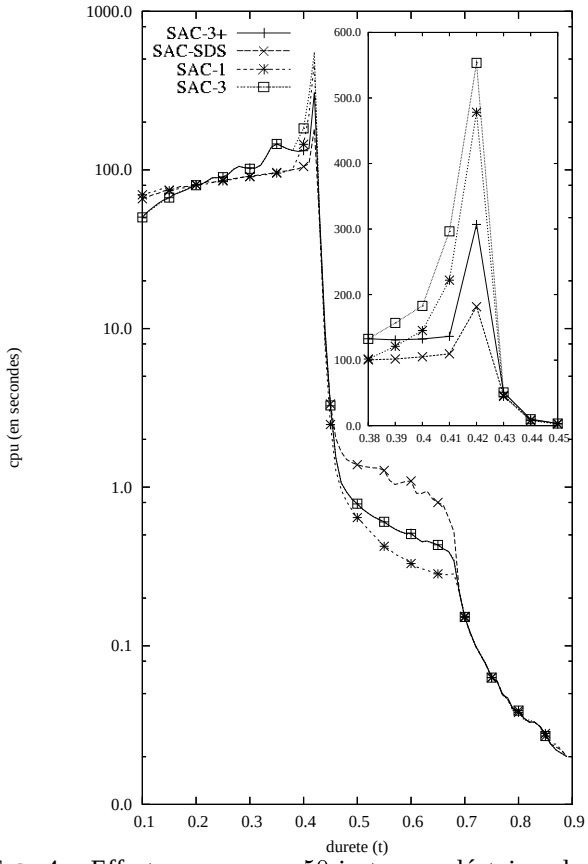

FIG. 4 – Effort moyen pour 50 instances aléatoires de la classe (100,20,1,t)

forme b*g au niveau du critère noté #brs.

Pour commencer, nous avons étudié les deux classes d'instances aléatoires binaires introduites dans [4]. La première classe (100,20,0.05,t) correspond à des réseaux de contraintes de faible densité ayant 100 variables, 20 valeurs par domaine et une densité de 0.05 (c'est-à-dire, 248 contraintes). La deuxième classe (100,20,1,t) correspond à des réseaux beaucoup plus denses (puisque complets) ayant 100 variables, 20 valeurs par domaine et une densité de 1 (c'est-à-dire, 4950 contraintes). t représente la dureté des contraintes, c'est-à-dire, la proportion de tuples non autorisés au niveau des relations associées aux contraintes.

Sur la figure 3, nous pouvons observer que lorsque $t < 0.6$ (le début de la transition de phase), SAC-3 et SAC-3+ ont le même comportement et surpassent SAC-1 et SAC-SDS. Au niveau de la transition de phase, SAC-SDS devient clairement la meilleure approche. Pour les réseaux plus denses, des résultats similaires ont été obtenus (voir figure 4), et au pic de difficulté, SAC-SDS (181 s) est à peu près trois fois plus efficace que SAC-1 (478 s) et SAC-3 (553 s) et deux fois plus efficace que SAC-3+ (307 s). Ceci n'est pas réellement une surprise puisque les instances générées n'ont aucune structure, ce qui correspond au pire des cas pour SAC-3 et SAC-3+, la longueur moyenne des branches construites étant très faible.

Ensuite, nous avons considéré les instances académiques suivantes :

- deux instances du problème de coloration d'échiquier [2], notés cc-20-2 et cc-20-3, comportant des contraintes quaternaires,
- deux instances "Golomb ruler"[3], notées gr-34-8 et gr-34-9, comportant des contraintes binaires et ternaires,
- deux instances "queen attacking"[4], notées qa-5 et qa-6, comportant uniquement des contraintes binaires.

La table 1 montre que SAC-3 et (plus précisément) SAC-3+ ont un meilleur comportement que SAC-1 et SAC-SDS sur les instances pour lesquelles la longueur moyenne des branches construites est relativement élevée. Par contre, pour les instances difficiles du Golomb ruler (instances ayant une structure très régulière), seul SAC-3+ donne des résultats sensiblement équivalents à ceux donnés par SAC-SDS.

Ensuite, nous avons testé des instances réelles, empruntées à l'archive FullRLFAP, correspondant à des instances du problème d'assignation de fréquences radios [8]. La table 2 montre les résultats obtenus sur quelques instances représentatives. Comme prévu, sur des instances qui sont déjà singleton arc consistantes ($scen02$, $graph14$), une amélioration importante est obtenue. Mais ce fait reste avéré pour d'autres instances contenant des parties sous-contraintes de taille importante (celles pour lesquelles la longueur moyenne des branches est élevée). Il apparaît clairement que, sur ce type d'instances structurées, utiliser SAC-3 ou SAC-3+ est la meilleure approche, d'autant plus que SAC-SDS nécessite trop de mémoire sur certaines de ces instances. Par ailleurs, il est intéressant de noter que SAC-3 et SAC-3+ sont plus rapides que SAC-1 et SAC-SDS sur certaines instances (voir, par exemple, cc-20-2 et $scen02$) alors que le nombre de tests singleton est similaire. Ceci est la conséquence de l'exploitation de l'incrémentalité de la consistance d'arc lors de la construction des branches.

Un autre point méritant d'être mentionné est que certaines solutions (apparaissant entre parenthèses après le temps cpu) ont été trouvées durant le pré-traitement de certaines instances. Par exemple, 16 solutions ont été trouvées pour l'instance $scen02$ à la fois par SAC-3 et par SAC-3+.

Finalement, nous avons testé quelques instances issues d'un problème d'ordonnancement. Nous avons sélectionné ici deux instances représentatives de l'ensemble des 60 instances "job shop" proposées par Sadeh et Fox [20]. Il est intéressant de noter (voir tableau 3) que SAC-3 et SAC-3+ sont plus efficaces pour résoudre ces instances que MAC lorsque celui-ci est exécuté de manière à trouver une solution. En effet, pour ces 2 instances, au moins une solution est trouvée de façon opportuniste lors des essais de

[3] Voir problem006 sur `http://4c.ucc.ie/~tw/csplib/`
[4] Voir problem029 sur `http://4c.ucc.ie/~tw/csplib/`

		SAC-1	SAC-SDS	SAC-3	SAC-3+
cc-20-2	cpu	14.44	14.43	3.46	3.48
(#X=0)	#scks	800	800	819	819
	#brs			21*38.0	21*38.0
cc-20-3	cpu	22.61	22.71	7.01	7.02
(#X=0)	#scks	1200	1200	1220	1220
	#brs			23*52.1	23*52.1
gr-34-8	cpu	66.08	17.43	38.28	18.62
(#X=351)	#scks	6335	3340	5299	1558
	#brs			1683*2.1	499*1.9
gr-34-9	cpu	111.44	31.29	91.50	32.03
(#X=513)	#scks	8474	4720	11017	2013
	#brs			3618*2.0	675*1.7
qa-5	cpu	2.47	2.50	.93	.96
(#X=9)	#scks	622	622	732	732
	#brs			127*4.8	127*4.8
qa-6	cpu	27.55	14.34	8.23	4.38
(#X=48)	#scks	2523	1702	2855	1448
	#brs			384*6.5	197*6.3

TAB. 1 – Instances académiques

		SAC-1	SAC-SDS	SAC-3	SAC-3+
$scen$02	cpu	20.97	20.73	4.09 (16 sols)	4.08 (16 sols)
(#X=0)	#scks	8004	8004	8005	8005
	#brs			53*151.0	53*151.0
$scen$05	cpu	11.79	20.03	1.55 (1 sol)	1.87
(#X=13814)	#scks	6513	4865	4241	2389
	#brs			164*24.1	126*16.7
$scen$11	cpu	112.22	-	25.89	26.48
(#X=0)	#scks	26856	-	26972	26972
	#brs			145*185.2	145*185.2
$graph$03	cpu	215.88	136.93	74.97	39.10
(#X=1274)	#scks	20075	17069	22279	8406
	#brs			2217*8.8	953*7.3
$graph$10	cpu	1389.59	-	675.64	349.53
(#X=2572)	#scks	74321	-	82503	29398
	#brs			8230*8.8	3111*8.1
$graph$14	cpu	154.16	-	31.17 (10 sols)	32.72 (10 sols)
(#X=0)	#scks	36716	-	36719	36719
	#brs			53*692.7	53*692.7

TAB. 2 – Instances RLFAP

		SAC-1	SAC-SDS	SAC-3	SAC-3+	MAC
$enddr$1-10-by-5-10	cpu	29.61	29.58	18.17 (4 sols)	19.04 (4 sols)	181.68
(#X=0)	#scks	5760	5760	6019	6019	-
	#brs			330*17.4	330*17.4	-
$enddr$2-10-by-5-2	cpu	40.40	38.58	31.13 (1 sol)	31.07 (1 sol)	211.52
(#X=0)	#scks	6315	6315	6718	6718	-
	#brs			466*13.5	466*13.5	-

TAB. 3 – Instances Job-Shop

recherche gloutonne par SAC-3 et SAC-3+. Par exemple, la première solution de l'instance $enddr1$-10-by-5-10 a été trouvée par SAC-3 et SAC-3+ en moins de 2 secondes (temps non indiqué dans le tableau). Ceci peut s'expliquer par les nombreux redémarrages inhérents à l'approche gloutonne.

8 Conclusion

Effectuer un effort conséquent afin d'atteindre un résultat concret est acceptable. Par exemple, établir la singleton consistance d'arc sur un réseau de contraintes peut être justifié si cela permet un certain niveau de filtrage. Au contraire, appliquer un algorithme SAC sur un réseau déjà singleton arc consistant est un problème puisque cela correspond purement et simplement à perdre du temps.

Les deux algorithmes, SAC-3 et SAC-3+, introduits dans cet article, peuvent être considérés comme apportant une réponse (partielle) à ce problème. En effet, lorsqu'une instance est sous-contrainte ou, plus généralement, contient des parties "faciles" de grande taille, comme cela peut se présenter avec des applications réelles, exploiter la recherche pendant l'inférence peut payer. Cela permet de réduire significativement le temps nécessaire à établir la singleton consistance d'arc, d'effectuer un apprentissage à partir des conflits (qui pourra en particulier être utilisé plus tard pendant la recherche) et de trouver potentiellement des solutions pendant l'inférence. Ceci a été confirmé par nos expérimentations. On notera également la très bonne complexité spatiale de SAC-3, ce qui le rend applicable sur des réseaux de contraintes de très grande taille.

Nous pensons que cette approche mérite de nouveaux approfondissements afin de déterminer, dans quelle mesure, maintenir la singleton consistance d'arc pendant la recherche, sur la base d'une approche gloutonne, pourrait être une alternative viable à MAC.

Remerciements

Cet article a reçu le soutien du programme COCOA de la Région Nord/Pas-de-Calais et celui de l'IUT de Lens.

Références

[1] R. Bartak. A new algorithm for singleton arc consistency. *Actes de FLAIRS'04*, 2004.

[2] M. Beresin, E. Levin, and J. Winn. A chessboard coloring problem. *The College Mathematics Journal*, 20(2) :106–114, 1989.

[3] C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency. *Actes de ECAI'04 workshop on modelling and solving problems with constraints*, pages 20–29, 2004.

[4] C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. *Actes de IJCAI'05*, à paraître, 2005.

[5] C. Bessière and J. Régin. Refining the basic constraint propagation algorithm. *Actes de IJCAI'01*, pages 309–315, 2001.

[6] L. Bordeaux, E. Monfroy, and F. Benhamou. Improved bounds on the complexity of kB-consistency. *Actes de IJCAI'01*, pages 303–308, 2001.

[7] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. *Actes de ECAI'04*, pages 146–150, 2004.

[8] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4(1) :79–89, 1999.

[9] A. Chmeiss and L. Sais. About the use of local consistency in solving CSPs. *Actes de ICTAI'00*, pages 104–107, 2000.

[10] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. *Actes de CP'97*, pages 312–326, 1997.

[11] R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. *Actes de IJCAI'97*, pages 412–417, 1997.

[12] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[13] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. *Actes de CP'03*, pages 480–494, 2003.

[14] O. Lhomme. Consistency techniques for numeric CSPs. *Actes de IJCAI'93*, pages 232–238, 1993.

[15] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[16] P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. *Actes de IPCO'96*, pages 389–403, 1996.

[17] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. *Actes de CP'00*, pages 353–368, 2000.

[18] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. *Actes de PPCPA'94*, 1994.

[19] D. Sabin and E. Freuder. Understanding and improving the MAC algorithm. *Actes de CP'97*, 1997.

[20] N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86 :1–41, 1996.

[21] Y. Zhang and R.H.C. Yap. Making AC3 an optimal algorithm. *Actes de IJCAI'01*, pages 316–321, 2001.